Solving the cart-pole problem with deep Q-learning

Bryan C. Chen, Raymond T. Salzmann

Abstract

The cart-pole problem is one which at its essence involves an agent balancing a free-moving pole on a cart. In the scenario where a human is the agent, we have the ability to see the angle and velocity of the pole and intuitively know how to move our "cart" to maintain the balance of the pole. However, in the cart-pole problem where we use a simulated frictionless cart as the agent, we must train the agent to adequately assess and move accordingly to maintain the balance of the pole. In order to train this simulated agent, we utilize a deep Q-learning model with experience replay to teach the agent how to move the cart through a number of training episodes, until the agent is able to adequately keep the pole balanced on the cart. We then utilized a self-made scoring method to quantitatively evaluate the performance of our agent. The environment used to simulate this experiment is from prior research; however, the final network, trained models, analysis of the results, and the entirety of this manuscript are wholly new.

1. Introduction

It is important to note the variables that are present within the cart-pole problem, and how they will play into our agent's learning and the above explained methodology. The variables apparent in the environment in the cart-pole problem are the cart position, cart velocity, pole velocity, and pole angle. Even though we know the variables that are present within the environment, since we can not necessarily model them, we must use a model-free form of reinforcement learning. When examining our approach and results to this cart-pole problem, it is important to first define two things at an elementary level, to have some knowledge of the underlying concepts we utilized to train our agent. First, it is important to understand the functionality and background of deep Q-learning. Q-learning is a model-free reinforcement learning algorithm with the intent of assessing the quality of an action, and ultimately to learn the policy that will maximize the agent's total reward. Because "reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value…function," we must find a way to prevent these instabilities (Mnih 2015). One of the ways to address these instabilities is with a form of Q-learning called deep Q-learning and the use of a deep Q-network, or DQN. DQNs use two key ideas. First, they use a "biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution." Second, DQNs use "an iterative update that adjusts the action-values…towards target values that are only periodically updated, thereby reducing correlations with the target." By using these two ideas, DQNs avoid the instabilities caused by "correlations present in the sequence of observations, the fact that small updates to *Q* may significantly change the policy and therefore change the data distribution, and the correlations between the action-values…and the target values." With this knowledge, we set out to utilize a deep Q-learning algorithm by developing a sequence of observations, actions, and rewards in order to properly train our agent. The goal of our agent in our DQN is to select actions which maximize the cumulative future reward for the agent. In more formal terms, we attempt approximate and maximize the optimal action-value function:

$$Q^*(s,a) = max_{\pi}\mathrm{E}(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi)$$

The above equation represents the maximum sum of rewards $\mathbf{r_t}$ discounted by the discount factor $\boldsymbol{\gamma}$ at each of the time steps **t**, which is achievable by a policy $\boldsymbol{\pi} = \mathbf{P(a|s)}$, after an observation **s** has been recognized by the agent and an action **a** has been chosen. The experience replay that we utilize in our application allows us to store our agent's experiences at each time step **t**, such that we can append the experiences to a data set, and thus be able to retroactively access the agent's experiences. This is important because during our agent's learning, we apply deep Q-learning

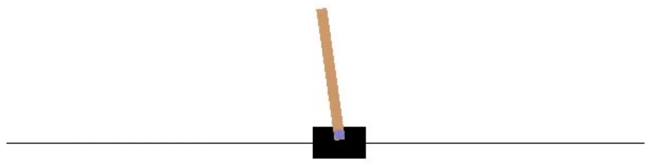


Figure 1. Illustration of a cart-pole.

updates to random samples of experiences drawn from the pool of stored samples in the data set mentioned above. From this we are able to calculate the loss function, which allows us to quantitatively analyze our agent's performance. Along with these hyperparameters gained from the experiences of the agent, we also have to decide on several values that allow for our agent to perform its best during training, including the discount factor γ, the batch size, the total episodes and epochs, and the learning rate for the agent. It is important to note the variables that are present within the cart-pole problem, and how they will play into our agent's learning and the above explained methodology. The discount factor γ determines how far we take future rewards into account, the batch size determines how much of our stored data to draw influence from when deciding actions, the total episodes and epochs per episode determines how long to train our model for and how over or underfit the resulting model will be, and the learning rate determines how quickly the model will learn. On top of all of that, we also have to determine how large and complex of a neural network we want to build. For example, a very complex model may be stronger than a simpler model, but it will take much longer to train and may also end up overfitting. With all of these hyperparameters taken into account, we were able to use our deep Q-learning algorithm, which is later explained in greater detail, to accurately train our agent to solve the cart-pole problem, thus maintaining the balance of the pole by constantly adjusting the position of the cart.

2. Background and Related Work

When first contemplating the idea of approaching one of the most well-known problems in the reinforcement learning sphere, we thought it would be a good idea to first get a background on the different algorithmic solutions that people have attempted in the past. It is important to know that, as explained in the introduction, all proposed solutions are model-free as it is realistically impossible to fully model the agent's environment. We first looked at a paper, "Comparison of Reinforcement Learning Algorithms applied to the Cart-Pole Problem," which proposed a number of possible reinforcement learning based approaches to the proposed problem. The first algorithm introduced is actually Q-learning, which is the method we have discussed in the introduction, and ultimately the algorithm we wound up using, so we will not go into as much detail about this here. However, it is important to note that based on the results presented in Nagendra *et.al,* we decided to use Q-learning as it was a reasonable algorithm to implement with just two people in a short amount of time, as well as one of the most efficient algorithms. By this we mean it is the most efficient in the number of episodes until optimal policy is found.

Amongst the Q-learning algorithms presented in the prior paragraph, we also took a look at using either policy gradient or actor-critic algorithms to try and solve our Cart-Pole problem. These algorithms would solve any possible issue should the value function approximation become too complex, due to their stochastic action selection at every step. However, these algorithms have a major drawback of complexity, especially in the case of the actor-critic algorithm. In the actor-critic algorithm, we utilize two separate networks, one for the action selection, and one for evaluation. The overall solution flowchart can be found in *Figure 2*, which will accurately show the two networks that are necessary to adequately implement the algorithm. With this in mind, these networks both consist of a single neuron, so while the implementation of two networks is complicated as a whole, the implementation of each is not overly complicated. In a similar vein, the policy gradient algorithm has a similar drawback, in that there is an extremely high variance observed, thus the method proposed is to combine the two algorithms to make an overall method that may fix a lot of the drawbacks of the Q-learning algorithm. However, with the complexity or implementation, and the amount of episodes needed to find the optimal policy being nearly three times that of Q-learning, we decided to not pursue a solution of this kind.

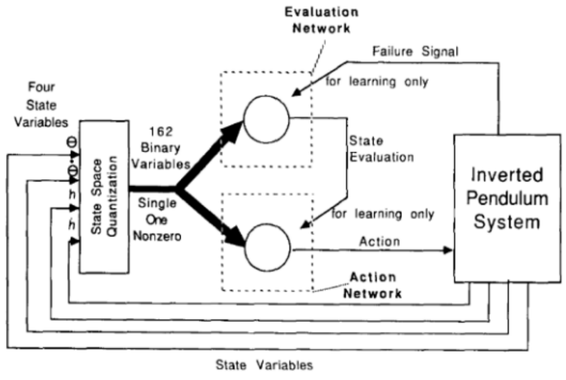


Figure 2. Flowchart of the actor-critic method.

Finally, we looked at a method formally known as Temporal Difference with Value Function Approximation. This method stems from the combination of two separate algorithms: value function approximation and stochastic gradient descent. While the other methods that we have discussed thus far have utilized only discretized state spaces, with this TD method, we are able to use a state space in which we would be able to use a continuous state space. While this solves the disadvantage of the formerly proposed methods, we are able to use a discretized state space, and thus the main advantage of this method is null. However, it is important to note on the topic of efficiency of a method, this TD with Value Function Approximation method actually found the optimal policy in only 19 episodes, the least amount of episodes needed to accurately find the optimal policy of all methods researched.

Ultimately, we decided to pursue a solution using the Q-learning algorithm presented and explained in the Introduction, as we believed it to be the most robust, efficient, and plausible solution to our Cart-Pole problem. Moving forward, all results and approaches discussed will solely be in explanation of our implementation of the Q-learning algorithm while utilizing deep neural networks.

3. Approach

In order to effectively solve the cart-pole problem, we must first understand the deep Q-learning algorithm. First, we define our action network $\mathbf{Q^A}$ with random weights as with any reinforcement learning problem, and we set our target network $\mathbf{Q^T}$ equal to our action network $\mathbf{Q^A}$ to begin. We also initialize our replay memory **D** with capacity **N**. Next, we enter the loop of the algorithm. For each episode **m** of **M** total episodes, initialize the episode reward to 0, reset the environment, and obtain the first observation $\mathbf{s_1}$. For each epoch **t** of **T** total epochs, pick a random action $\mathbf{a_t}$ with probability **ε**, otherwise pick action $\mathbf{a_t}$ equal to $\mathbf{max_a Q^A(s_t, a)}$ with probability **ε - 1**. Execute the action $\mathbf{a_t}$, observe the reward $\mathbf{r_{t+1}}$ and state $\mathbf{s_{t+1}}$, and add the reward to the cumulative episode reward. Store the transition matrix $\mathbf{(s_t, a_t, r_{t+1}, s_{t+1})}$ into the replay memory and remove old memory if needed. Next, we train the network using the stored transitions. We sample random minibatches of transitions from **D** and perform batch gradient descent on $\mathbf{Q^A}$ with $\mathbf{L = F(Q^A(s_t, a_t) - [r_t + \gamma max_a Q^T(s_{t+1}, a)])}$. We then set our state $\mathbf{s_t}$ to our next state $\mathbf{s_{t+1}}$ and if this next state is terminal, end this episode **m**. After **T** total epochs, we report the cumulative episode reward and save the model of $\mathbf{Q^A}$ if the reward is higher than a certain threshold **R**. Lastly, we periodically set the target network $\mathbf{Q^T}$ to our action network $\mathbf{Q^A}$.

Algorithm 1 Deep Q-learning with experience replay

```
Initialize: action network $Q^A$ with random weights; target network $Q^T = Q^A$; replay memory D with capacity N
for episode $m$ = 1,M do
  Initialize: episode_reward = 0
  Reset environment and obtain first observation $s_1$
  for $t$ = 1,T do
    With probability ε, $a_t$ = random; otherwise $a_t = \max_a Q^A(s_t, a)$
    Execute $a_t$ and observe $r_{t+1}$, $s_{t+1}$
    episode_reward += $r_{t+1}$
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in D and remove old data if needed
    Train network:
      Sample random minibatch of transitions from D
      Perform batch GD on $Q^A$ with $L = F(Q^A(s_t, a_t) - [r_t + \gamma \max_a Q^T(s_{t+1}, a)])$
    Set $s_t = s_{t+1}$
    If $s_{t+1}$ is terminal: break
  end for
  Report episode_reward and save model of $Q^A$ if reward is higher than threshold R
  Every $k$ episodes: set target network $Q^T = Q^A$
end for
```

In order to make our model as practical and available as possible, we decided to limit our machines to train using only the CPU and not the GPU. Tuning the hyperparameters is certainly required, as the resulting models' behaviors will vary greatly depending on initial conditions. As such, we will discuss only the results of our best model, even though we discovered several that solved the cart-pole problem. The metrics for what we determine to be the "best model" are based on how much the cart moves back and forth in order to balance the pole and for how long it balances the pole.

4. Results

The results of this experiment were largely successful. We set out to see if we could teach a cart in simulation to balance a pole using Q-learning techniques and were able to do so. After spending much time tuning parameters and training and testing hundreds of models, we were finally able to teach the cart how to balance the pole for quite a while, probably indefinitely. While we were initially concerned with the complexity of the model and the length of the training time, we decided to take the Occam's razor approach and simplify our model. Our resulting fully-connected model consists of only two hidden layers containing 256 nodes each, and all layers use a rectified linear activation function. The final layer outputs one of 3

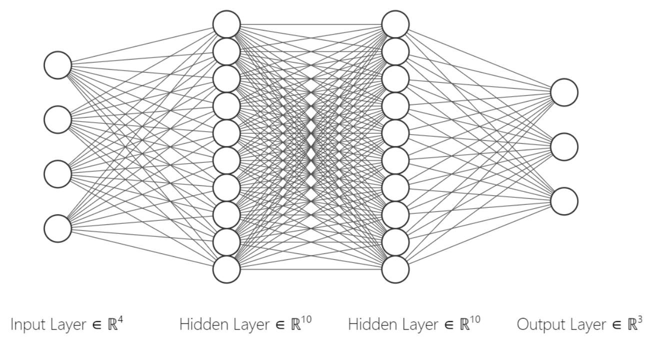


Figure 3. Example of a fully-connected deep neural network. In our case, each hidden layer has 256 nodes instead of 10.

actions: move left, move right, or don't move at all. Previous efforts yielded neural networks with at least three hidden layers and combinations of up to 1024 nodes per hidden layer. Not only did this model take too much time to train on our machines (over 2 hours), but it became highly susceptible to variations in the other hyperparameters, specifically the learning rate and decay values. This suggested that our previous models were overfitting to our training data and were not generalizable enough to our OpenAI gym simulation.

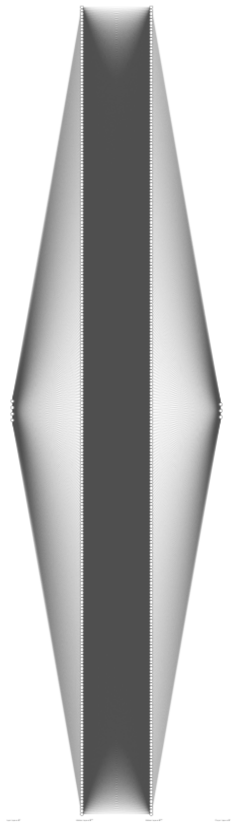


Figure 4. Our actual network zoomed out to show all nodes.

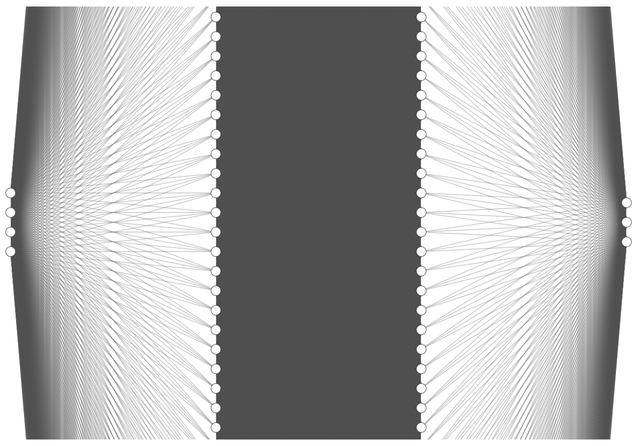
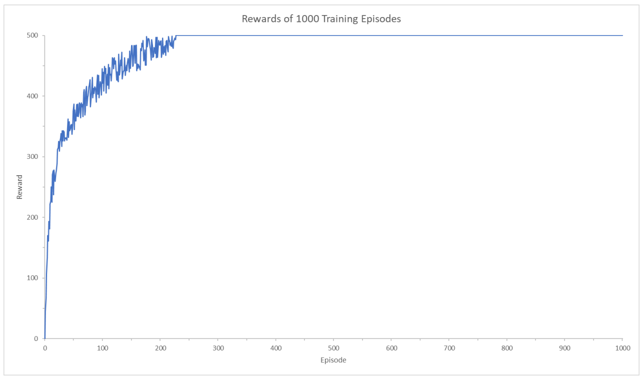


Figure 5. Our network zoomed in to show fully-connectedness.

Our finalized model performed well even with a training time of around 4 minutes. It would have been impossible to optimize the rest of the hyperparameters without utilizing an extensive grid search, but the hyperparameters that we used were sufficient for properly balancing the pole for at least 2 minutes. We trained our model a total of 1000 episodes and 500 epochs per episode using a batch size of 32, a gamma value of 0.999, a learning rate of 0.0001, and an epsilon value that begins at 1.0 and decays by a factor of 0.95 until a minimum of 0.01. The replay buffer has a capacity of 10,000, which proved to be more than enough to solve the cart-pole problem. Lastly, we trained the model using the Adam optimizer and evaluated its performance using the mean squared error loss criterion. During the training phase, we saved a model of each episode if its reward is higher than a reward threshold, which we arbitrarily set to 500, and set our target network to our action network every 3 episodes, which we also picked arbitrarily. From the saved models, we tested each in descending order beginning with the model with the highest reward against our simulation. One interesting observation is that the most recent model with the highest reward did not perform the best, and it was our 6th most recent model (i.e. model 995 out of 1000) that performed the best, which is indicative of the fact that the model with the high rewards may be slightly overfit, but overall, there were numerous models (over 700 in fact) that achieved our target reward goal of solving the cart-pole problem. We only tested the last 10, but based on their performance, we can reasonably conclude that all 773 models will balance the pole for at least several minutes. Out of the 10 models we tested, it was with model 995 that the cart moved back and forth the least to balance the pole.



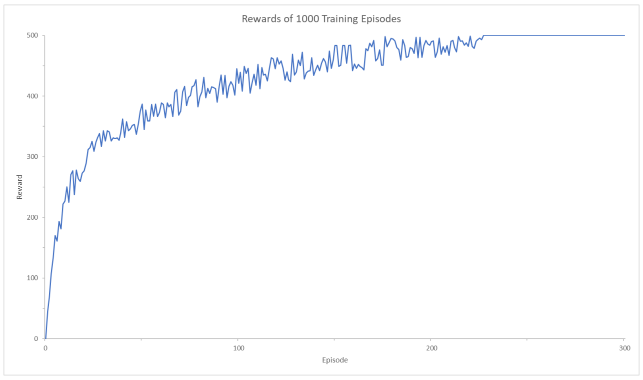


Figure 6. Top: Rewards of all 1000 training episodes. Bottom: Rewards of the first 300 training episodes. Every episode after the 226th achieved our target reward of 500.

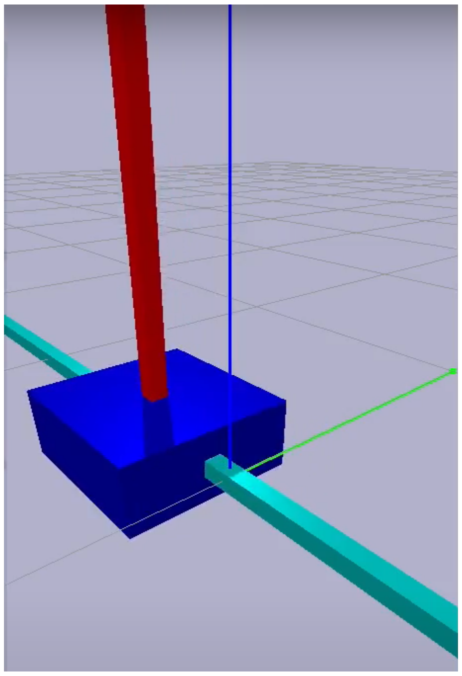


Figure 7. The end of one run of our cart-pole simulation using model 995. Notice the cart-pole is on the left of the z-axis.

5. Conclusion

Within the simulation, the cart performed just as intended: the cart successfully balanced the pole on the infinitely long track it was given. However, its performance in the simulation has some real-world implications. While the 6th model documented in this manuscript is our best-performing model, it is by no means perfect. During the simulation it can be seen that the cart slowly shifts to the left over the course of several minutes while it is balancing the pole. While this is acceptable in simulation, if this model were to be implemented on a real-life cart for a practical demonstration, the cart would definitely fall off the left side of the track after some time. This indicates the model is either not aggressive enough in keeping the pole upright, is not quick enough to react to the toppling pole, or is not somehow trained enough with catching up to the pole from the right rather than from the left, akin to catching a falling object with one's non-dominant hand.

Future experiments on the cart-pole problem would need to take into account the length of the track in order to be feasibly implemented in a real demonstration, and several behaviors should also be explored. One such behavior could be an aggressive cart that quickly moves back and forth, changing the direction of the pole quickly enough to effectively cheat gravity and give the pole zero time to fall over since its center of gravity will always remain between the two turning points of the cart. An even more interesting behavior could be a more sophisticated model that moves the cart at a specific velocity to a specific position such that the pole is near-perfectly balanced and thus no longer requires any further adjustment from the cart, eliminating the need for the cart to move rapidly back and forth. We did see these two behaviors from our other models, but its performance was quite lackluster after the first few seconds and the pole did indeed end up toppling over within the first minute of simulation.

Given enough time and computing power, models exhibiting both of these behaviors that are able to balance the pole indefinitely can surely be found. One such method of developing these models could be to combine multiple reinforcement learning methods together, rather than using only DQN, for example.

References

Brockman, G., Cheung, V., Pettersson, L. *et al*. OpenAI Gym. 2016. arXiv:1606.01540.

LeNail, A. NN-SVG: Publication-Ready Neural Network Architecture Schematics. Journal of Open Source Software, 4(33), 747, 2019. https://doi.org/10.21105/joss.00747.

Mnih, V., Kavukcuoglu, K., Silver, D. *et al*. Human-level control through deep reinforcement learning. Nature 518, 529-533, 2015. https://doi.org/10.1038/nature14236.

Nagendra, S., Podila, N., Ugarakhod R. *et al*. Comparison of Reinforcement Learning algorithms applied to the Cart Pole problem. International Conference on Advances in Computing, Communications and Informatics, 2017. arXiv:1810.01940.